Unlink <modname> Usage : Unlinks module(s) from memory @WCREATE Syntax: Wcreate [opt] or /wX [-s=type] xpos ypos xsiz ysiz fcol bcol [bord] Usage : Initialize and create windows Opts : -? = display help -z = read command lines from stdin -s=type = set screen type for a window on a new screen @XMODE Syntax: XMode <devname> [params]

# AUSTRALIAN OS9 NEWSLETTER

Usage : Displays or changes theparameters of an SCF type device @COCOPR Syntax: cocopr [<opts>] {<path> [<opts>]} Function: display file in specified format gets defaults from /dd/sys/env.file Options : -c set columns per page -f use form feed for trailer -h=num set number of lines after header -l=num set line length -m=num set left margin -n=num set starting line number and incr -o truncate lines longer than lnlen -p=num set number of lines per page -t=num number of lines in trailer -u do not use title

| | | |
|---|---|---|
| EDITOR | Gordon Bentzen | (07) 344-3881 |
| SUB-EDITOR | Bob Devries | (07) 372-7816 |
| TREASURER | Don Berrie | (07) 375-1284 |
| LIBRARIAN | Jean-Pierre Jacquet | (07) 372-4675 |
| | Fax Messages | (07) 372-8325 |
| SUPPORT | Brisbane OS9 Users Group | |

-u=title use specfied title -x=num set starting page number -z[=path] read file names from stdin or <path> if given @CONTROL Syntax: control [-e] Usage : Control Panel to set palettes, mouse and keyboard parameters and monitor type for Multi-Vue. Selectable from desk utilities menu as the Control Panel. Opts : -e = execute the environment file @GCLOCK Syntax: gclock Usage : Alarm clock utility for Multi-Vue. Selectable from desk utilities menu as Clock. @GCALC Syntax: gcalc Usage : Graphics calculator utility for Multi-Vue. Selectable form desk utilities menu as

## ADDRESSES

**Editorial Material:**
Gordon Bentzen
8 Odin Street
SUNNYBANK Qld 4109

**Library Requests:**
Jean-Pierre Jacquet
27 Hampton Street
DURACK Qld 4077

## CONTENTS

Volume 6 December 1992 Number 11

Calculator. @GCAL Syntax: gcal Usage : Calendar/Memo book utility for Multi-Vue. Selectable as Calendar from the desk utilities menu. @GPRINT

**AUSTRALIAN OS9 NEWSLETTER**
**Newsletter of the National OS9 User Group**
**Volume 6 Number 11**

**EDITOR** : Gordon Bentzen
**SUBEDITOR** : Bob Devries
**TREASURER** : Don Berrie
**LIBRARIAN** : Jean-Pierre Jacquet
**SUPPORT** : Brisbane OS9 Level 2 Users Group.

## SEASONS GREETINGS

Yes it is that time of year again when many of us prepare for Christmas celebrations in one way or another. Perhaps the time of year will mean a welcome break from studies or work, a well earned holiday, a time to share with your family or maybe like me, it will be just more work as usual.

In whatever way you plan to spend the time, we wish you a very merry Christmas and a happy new year - from Bob Devries, Don Berrie, Jean-Pierre Jacquet and Gordon Bentzen.

We have gained quite a few new members in the last year and we again welcome you and thank you for your interest. We are most grateful for the support of those members who have renewed subscriptions this year. Some of our long term supporters have moved away from the CoCo or OS-9 and have continued to subscribe. Our special thanks to you.

## JANUARY / FEBRUARY NEWSLETTER

This is a good time to remind you that at this time of year, we too take a little holiday from newsletter production. This December issue will be the last until early February 1993. A newsletter WILL NOT be sent out in January.

## COCOFEST- MELBOURNE

In the editorial last month I presented some comments on the CoCoFest held in Melbourne 24th & 25th October 1992 and this month we have a couple of shots which were taken from my video and digitized.

Fred Remin leads a discussion

## A new member writes :-

"These days I find that 99% of the time I use OS-9 rather than Tandy Basic, and I like the windowing capabilities of OS-9. I would also like to upgrade to a hard drive for my CoCo, but need some expert advice as to what to buy that would be compatible with our Australian electrical standards.

My hardware set-up consists of a CoCo3 with 512k, Multi-pak interface (with CoCo3 upgrade), two floppy drives (FD-502), CM-8 Colour Monitor and a DMP-200 Printer."

## Solution :-

Given the available hardware, as above, probably the easiest and cheapest way to a CoCo Hard Drive would be the **Burke & Burke XTC** system. The Burke & Burke system uses their **interface** to adapt an IBM type drive and controller.

A standard IBM or clone type hard drive of say 20, 30 or 40meg would be a good choice. A quarter card controller of MFM or RLL type would also be selected. This system will also allow the use of the Burke & Burke XT-ROM to allow booting directly from the hard drive. Some of this equipment is becoming hard to find now, as the IBM and clone world have commonly gone to IDE drives and controllers.

Our November newsletter included a discussion on the IDE drives for the CoCo and it seems that this could be a real option if Burke & Burke do release suitable drivers. In the meantime we would be glad to hear from any member who is able to advise of a source for the MFM and RLL drives and controllers.

Cheers, Gordon.

L to R Don Berrie, Andrew Donaldson, Bob Devries

# A C Tutorial
# Chapter 11 - Structures and Unions

**WHAT IS A STRUCTURE?**
A structure is a user defined data type. You have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a combination of several different previously defined data types, including other structures we have defined. An easy to understand definition is, a structure is a grouping of related data in a way convenient to the programmer or user of the program.

The best way to understand a structure is to look at an example, so if you will load and display STRUCT1.C, we will do just that. The program begins with a structure definition. The key word "struct" is followed by some simple variables between the braces, which are the components of the structure. After the closing brace, you will find two variables listed, namely "boy", and "girl".

According to the definition of a structure, "boy" is now a variable composed of three elements, "initial", "age", and "grade". Each of the three fields are associated with "boy", and each can store a variable of its respective type. The variable "girl" is also a variable containing three fields with the same names as those of "boy" but are actually different variables. We have therefore defined 6 simple variables.

**A SINGLE COMPOUND VARIABLE**
Let's examine the variable "boy" more closely. As stated above, each of the three elements of "boy" are simple variables and can be used anywhere in a C program where a variable of their type can be used. For example, the "age" element is an integer variable and can therefore be used anywhere in a C program where it is legal to use an integer variable, in calculations, as a counter, in I/O operations, etc.

The only problem we have is defining how to use the simple variable "age" which is a part of the compound variable "boy". We use both names with a decimal point between them with the major name first. Thus "boy.age" is the complete variable name for the "age" field of "boy". This construct can be used anywhere in a C program that it is desired to refer to this field. In fact, it is illegal to use the name "boy" or "age" alone because they are only partial definitions of the complete field. Alone, the names refer to nothing.

**ASSIGNING VALUES TO THE VARIABLES**
Using the above definition, we can assign a value to each of the three fields of "boy" and each of the three fields of "girl". Note carefully that "boy.initial" is actually a "char" type variable, because it was assigned that in the structure, so it must be assigned a character of data. Notice that "boy.initial" is assigned the character 'R' in agreement with the above rules. The remaining two fields of "boy" are assigned values in accordance with their respective types. Finally the three fields of girl are assigned values but in a different order to illustrate that the order of assignment is not critical.

**HOW DO WE USE THE RESULTING DATA?**
Now that we have assigned values to the six simple variables, we can do anything we desire with them. In order to keep this first example simple, we will simply print out the values to see if they really do exist as assigned. If you carefully inspect the "printf" statements, you will see that there is nothing special about them. The compound name of each variable is specified because that is the only valid name by which we can refer to these variables. Structures are a very useful method of grouping data together in order to make a program easier to write and understand.

This first example is too simple to give you even a hint of the value of using structures, but continue on through these lessons and eventually you will see the value of using structures. Compile and run STRUCT1.C and observe the output.

**AN ARRAY OF STRUCTURES**
Load and display the next program named STRUCT2.C. This program contains the same structure definition as before but this time we define an array of 12 variables named "kids". This program therefore contains 12 times 3 = 36 simple variables, each of which can store one item of data provided that it is of the correct type. We also define a simple variable named "index" for use in the for loops.

In order to assign each of the fields a value, we use a for loop and each pass through the loop results in assigning a value to three of the fields. One pass through the loop assigns all of the values for one of the "kids". This would not be a very useful way to assign data in a real situation, but a loop could read the data in from a file and store it in the correct fields. You might consider this the crude beginning of a data base, which it is. In the next few instructions of the program we assign new values to some of the fields to illustrate the method used to accomplish this. It should be self explanatory, so no additional comments will be given.

**A NOTE TO PASCAL PROGRAMMERS**
Pascal allows you to copy an entire RECORD with one statement. This is not possible in C. You must copy each element of a structure one at a time. As improvements to the language are defined, this will be one of the refinements. In fact, some of the newer compilers already allow structure assignment. Check your compiler documentation to see if your compiler has this feature yet.

**WE FINALLY DISPLAY ALL OF THE RESULTS**
The last few statements contain a for loop in which all of the generated values are displayed in a formatted list. Compile and run the program to see if it does what you expect it to do.

**USING POINTERS AND STRUCTURES TOGETHER**
Load and display the file named STRUCT3.C for an example of using pointers with structures. This program is identical to the last program except that it uses pointers for some of the operations. The first difference shows up in the definition of variables following the structure definition. In this program we define a pointer named "point" which is defined as a pointer that points to the structure.

It would be illegal to try to use this pointer to point to any other variable type. There is a very definite reason for this restriction in C as we have alluded to earlier and will review in the next few paragraphs. The next difference is in the for loop where we use the pointer for accessing the data fields. Since "kids" is a pointer variable that points to the structure, we can define "point" in terms of "kids". The variable "kids" is a constant so it cannot be changed in value, but "point" is a pointer variable and can be assigned any value consistent with its being required to point to the structure. If we assign the value of "kids" to "point" then it should be clear that it will point to the first element of the array, a structure containing three fields.

**POINTER ARITHMETIC**
Adding 1 to "point" will now cause it to point to the second field of the array because of the way pointers are handled in C. The system knows that the structure contains three variables and it knows how many memory elements are required to store the complete structure. Therefore if we tell it to add one to the pointer, it will actually add the number of memory elements required to get to the next element of the array.

If, for example, we were to add 4 to the pointer, it would advance the value of the pointer 4 times the size of the structure, resulting in it pointing 4 elements farther along the array. This is the reason a pointer cannot be used to point to any data type other than the one for which it was defined. Now to return to the program displayed on your monitor. It should be clear from the previous discussion that as we go through the loop, the pointer will point to the beginning of one of the array elements each time. We can therefore use the pointer to reference the various elements of the structure.

Referring to the elements of a structure with a pointer occurs so often in C that a special method of doing that was devised. Using "point->initial" is the same as using "(*point).initial" which is really the way we did it in the last two programs. Remember that *point is the data to which the pointer points and the construct should be clear. The "->" is made up of the minus sign and the greater than sign. Since the pointer points to the structure, we must once again define which of the elements we wish to refer to each time we use one of the elements of the structure.

There are, as we have seen, several different methods of referring to the members of the structure, and in the for loop used for output at the end of the program, we use three different methods. This would be considered very poor programming practice, but is done this way here to illustrate to you that they all lead to the same result. This program will probably require some study on your part to fully understand, but it will be worth your time and effort to grasp these principles. Compile and run this program.

**NESTED AND NAMED STRUCTURES**
Load and display the file named NESTED.C for an example of a nested structure. The structures we have seen so far have been very simple, although useful. It is possible to define structures containing dozens and even hundreds or thousands of elements but it would be to the programmers advantage not to define all of the elements at one pass but rather to use a hierarchical structure of definition. This will be illustrated with the program on your monitor.

The first structure contains three elements but is followed by no variable name. We therefore have not defined any variables only a structure, but since we have included a name at the beginning of the structure, the structure is named "person". The name "person" can be used to refer to the structure but not to any variable of this structure type. It is therefore a new type that we have defined, and we can use the new type in nearly the same way we use "int", "char", or any other types that exist in C. The only restriction is that this new name must always be associated with the reserved word "struct".

The next structure definition contains three fields with the middle field being the previously defined structure which we named "person". The variable which has the type of "person" is named "descrip". So the new structure contains two simple variables, "grade" and a string named "lunch[25]", and the structure named "descrip". Since "descrip" contains three variables, the new structure actually contains 5 variables. This structure is also given a name "alldat", which is another type definition. Finally we define an array of 53 variables each with the structure defined by "alldat", and each with the name "student". If that is clear, you will see that we have defined a total of 53 times 5 variables, each of which is capable of storing a value.

**TWO MORE VARIABLES**
Since we have a new type definition we can use it to define two more variables. The variables "teacher" and "sub" are defined in the next statement to be variables of the type "alldat", so that each of these two variables contain 5 fields which can store data.

**NOW TO USE SOME OF THE FIELDS**
In the next five lines of the program, we will assign values to each of the fields of "teacher". The first field is the "grade" field and is handled just like the other structures we have studied because it is not part of the nested structure. Next we wish to assign a value to her age which is part of the nested structure. To address this field we start with the variable name "teacher" to which we append the name of the group "descrip", and then we must define which field of the nested structure we are interested in, so we append the name "age".

The teacher's status is handled in exactly the same manner as her age, but the last two fields are assigned strings using the string copy "strcpy" function which must be used for string assignment. Notice that the variable names in the "strcpy" function are still variable names even though they are made up of several parts each. The variable "sub" is assigned nonsense values in much the same way, but in a different order since they do not have to occur in any required order.

Finally, a few of the "student" variables are assigned values for illustrative purposes and the program ends. None of the values are printed for illustration since several were printed in the last examples. Compile and run this program, but when you run it you may get a "stack overflow" error. C uses it's own internal stack to store the automatic variables on but most C compilers use only a 2048 byte stack as a default.

This program has more than that in the defined structures so it will be necessary for you to increase the stack size. The method for doing this for some compilers is given in the accompanying COMPILER.DOC file with this tutorial. Consult your compiler documentation for details about your compiler. There is another way around this problem, and that is to move the structure definitions outside of the program where they will be external variables and therefore static. The result is that they will not be kept on the internal stack and the stack will therefore not overflow. It would be good for you to try both methods of fixing this problem.

**MORE ABOUT STRUCTURES**
It is possible to continue nesting structures until you get totally confused. If you define them properly, the computer will not get confused because there is no stated limit as to how many levels of nesting are allowed. There is probably a practical limit of three beyond which you will get confused, but the language has no limit. In addition to nesting, you can include as many structures as you desire in any level of structures, such as defining another structure prior to "alldat" and using it in "alldat" in addition to using "person".

The structure named "person" could be included in "alldat" two or more times if desired, as could pointers to it. Structures can contain arrays of other structures which in turn can contain arrays of simple types or other structures. It can go on and on until you lose all reason to continue. I am only trying to illustrate to you that structures are very valuable and you will find them great aids to programming if you use them wisely. Be conservative at first, and get bolder as you gain experience. More complex structures will not be illustrated here, but you will find examples of additional structures in the example programs included in the last chapter of this tutorial. For example, see the "#include" file "STRUCT.DEF".

**WHAT ARE UNIONS?**

Load the file named UNION1.C for an example of a union. Simply stated, a union allows you a way to look at the same data with different types, or to use the same data with different names. Examine the program on your monitor. In this example we have two elements to the union, the first part being the integer named "value", which is stored as a two byte variable somewhere in the computers memory. The second element is made up of two character variables named "first" and "second".

These two variables are stored in the same storage locations that "value" is stored in, because that is what a union does. A union allows you to store different types of data in the same physical storage locations. In this case, you could put an integer number in "value", then retrieve it in its two halves by getting each half using the two names "first" and "second". This technique is often used to pack data bytes together when you are, for example, combining bytes to be used in the registers of the microprocessor. Accessing the fields of the union are very similar to accessing the fields of a structure and will be left to you to determine by studying the example.

One additional note must be given here about the program. When it is run using most compilers, the data will be displayed with two leading f's due to the hexadecimal output promoting the char type variables to int and extending the sign bit to the left. Converting the char type data fields to int type fields prior to display should remove the leading f's from your display. This will involve defining two new int type variables and assigning the char type variables to them. This will be left as an exercise for you. Note that the same problem will come up in a few of the later files also.

Compile and run this program and observe that the data is read out as an "int" and as two "char" variables. The "char" variables are reversed in order because of the way an "int" variable is stored internally in your computer. Don't worry about this. It is not a problem but it can be a very interesting area of study if you are so inclined.

**ANOTHER UNION EXAMPLE**

Load and display the file named UNION2.C for another example of a union, one which is much more common. Suppose you wished to build a large database including information on many types of vehicles. It would be silly to include the number of propellers on a car, or the number of tires on a boat. In order to keep all pertinent data, however, you would need those data points for their proper types of vehicles.

In order to build an efficient data base, you would need several different types of data for each vehicle, some of which would be common, and some of which would be different. That is exactly what we are doing in the example program on your monitor. In this program, we will define a complete structure, then decide which of the various types can go into it. We will start at the top and work our way down. First, we define a few constants with the #defines, and begin the program itself. We define a structure named "automobile" containing several fields which you should have no trouble recognizing, but we define no variables at this time.

**A NEW CONCEPT, THE TYPEDEF**

Next we define a new type of data with a "typedef". This defines a complete new type that can be used in the same way that "int" or "char" can be used. Notice that the structure has no name, but at the end where there would normally be a variable name there is the name "BOATDEF". We now have a new type, "BOATDEF", that can be used to define a structure anyplace we would like to.

Notice that this does not define any variables, only a new type definition. Capitalizing the name is a personal preference only and is not a C standard. It makes the "typedef" look different from a variable name. We finally come to the big structure that defines our data using the building blocks already defined above. The structure is composed of 5 parts, two simple variables named "vehicle" and "weight", followed by the union, and finally the last two simple variables named "value" and "owner". Of course the union is what we need to look at carefully here, so focus on it for the moment.

You will notice that it is composed of four parts, the first part being the variable "car" which is a structure that we defined previously. The second part is a variable named "boat" which is a structure of the type "BOATDEF" previously defined. The third part of the union is the variable "airplane" which is a structure defined in place in the union. Finally we come to the last part of the union, the variable named "ship" which is another structure of the type "BOATDEF". I hope it is obvious to you that all four could have been defined in any of the three ways shown, but the three different methods were used to show you that any could be used. In practice, the clearest definition would probably have occurred by using the "typedef" for each of the parts.

**WHAT DO WE HAVE NOW?**
We now have a structure that can be used to store any of four different kinds of data structures. The size of every record will be the size of that record containing the largest union. In this case part 1 is the largest union because it is composed of three integers, the others being composed of an integer and a character each. The first member of this union would therefore determine the size of all structures of this type.

The resulting structure can be used to store any of the four types of data, but it is up to the programmer to keep track of what is stored in each variable of this type. The variable "vehicle" was designed into this structure to keep track of the type of vehicle stored here. The four defines at the top of the page were designed to be used as indicators to be stored in the variable "vehicle".

A few examples of how to use the resulting structure are given in the next few lines of the program. Some of the variables are defined and a few of them are printed out for illustrative purposes. The union is not used too frequently, and almost never by beginning programmers. You will encounter it occasionally so it is worth your effort to at least know what it is. You do not need to know the details of it at this time, so don't spend too much time studying it. When you do have a need for a variant structure, a union, you can learn it at that time. For your own benefit, however, do not slight the structure. You should use the structure often.

**PROGRAMMING EXERCISES**
1. Define a named structure containing a string field for a name, an integer for feet, and another for arms. Use the new type to define an array of about 6 items. Fill the fields with data and print them out as follows. A human being has 2 legs and 2 arms. A dog has 4 legs and 0 arms. A television set has 4 legs and 0 arms. A chair has 4 legs and 2 arms. etc.

2. Rewrite exercise 1 using a pointer to print the data out.

oooooooooooOOOOOOOOOOOOoooooooooo

## Programming for new platforms
## (Amiga-4000, Atari Falcon 030 and Kix/30)
By Michael Kearney

First thing, I am not a "programmer", (yet). I am more the hardware type, but am trying to learn C. I am writing this as personal opinion only and I hope it doesn't offend anyone.

Anyway,on with the show .................

After working in television for 13 years, I've decided to start my own video/ audio/computer services company. I will use a coupla 486 computers for voice mail and some other things. The audio/video computers are causing me to have brain spasms. I can't afford a Cray, Silicon Graphics or Sun (yet!) so I'm looking at the Amiga-4000, the Atari Falcon 030 and FHL Kix/30. Now as I understand it, the Kix/30 is Microware 68k compatible "right out of the box".

From all the stuff I've read about the others (A LOT!) there was a "rumor" that the latest version (2.4?) of OSK was being ported to the Atari Falcon. And according to the Amiga literature the CPU in the 4000 is "Motorola 68040 compatible". Now I was thinking (dangerous habit_don't try it unless you are willing to accept the consequences!) that if all the folk that had a hand in the OS9-LV2 upgrade could convince Microware to port OSK over to the Amiga_4000 and the Atari Falcon 030, Microware, along with it's programmers, could grab a share of an international market.

Think about it. For the first time (that I can remember) computers that already have a market, hardware-wise (Ataris, Amigas) are closer to being able to run OSK "at power up" than ever before. With the expertise that some of you have with OS9/K, combined with the hardware of these new machines (browse the Amiga and Atari forums and read the specs on the Amiga_4000 and Falcon 030) things could get interesting in the home computer market. Just think about it.

With Big Blue on the way out/down, there will be a gap that needs to be filled. There are some die hard CoCo OS9'ers out there (like myself) that don't really want to give up OS9, but want more computing power.

But before I spend $2600 or $3000 on a new computer system, I want to know that I will have hardware support! And with these new machines being released WITH FCC Class B, if I can run OSK on either of them I might be tempted to BUY more than one. And it sure would be nice to be able to run programs on either. Again, I am not a programmer, BUT if I were, I would think long and hard about this one.

After all, are you programming just to make other people happy? Or to make money/make yourself happy and then make other people happy, in that order?

These thoughts, ramblings and stuff are my own opinions. I hope I do not offend anyone. But I was just thinking about how home computers have changed and have become business/home computers.

The CPUs have been narrowed down to basically two; (_68xxxs and 386/486s). Operating Systems have been narrowed down to a few (Dos, OS/2, U*ix). OSK has a place in there. Especially since the new hardware is being designed to integrate audio/video/voice recognition. OSK should fit quite well in that type of hardware environment.

This could be the OSK opportunity of a lifetime.

oooooooooo0000000000oooooooooo

## NITROS9

```
 From: Alan Dekok  Date: 11-14-92 09:02
   To: All         Msg#: 7
Subj.: NitrOS9
 Area: OS9
```

Well, I finally got a copy of NitrOS9, and installed it on my system last night. After about an hour of fiddling in order to get a boot disk with modules that it liked (floppy systems....), it took roughly 20 minutes for the whole installation to go. Then I re-booted and rooted around to see what neat new toys I had.

It includes: most bug fixes (I didn't check for _all_ of them), 80x25 windows, enhanced grfdrv, and MUCH faster screen updates. If you ever had reason to complain about the slowness of OS9 screens,(and have NitrOS9) then SHUT UP! It's not a problem anymore <g>. Scrolling was about 35% faster, and things like PROC and PMAP just went <foom> onto the screen, whereas before you could actually see them print out the individual numbers and lines.

I tried out a bunch of programs, and everything seems to work the same as before. The docs included are quite straightforward and idiot proof (heck, even _I_ understood them). If you can make a new boot disk, installing this package is not much more complicated than that.

The only thing I didn't like, was some programs (games) expect a (40/80) by 24 screen, and the patches gave a (40/80)x25. This resulted in an extra line of junk at the bottom of the screen, but this would happen anytime you extended the screen, and there's not much you can do about it.

Soo... I'm satisfied. It works, it didn't crash on me, and it's FAST.

Exactly what I was looking for!

Alan DeKok.

oooooooooo0000000000oooooooooo

## 2 Meg Upgrade

by Marty Goodman

When Tony Di Stefano designed his 1 meg upgrade for the CoCo 3, he had HOPED that he could eventually lay hands on 4 bit wide by 256K DRAM chips that could be refreshed with a 256 cycle refresh, allowing them to be used with the GIME chip's memory manager. With those chips, one could build a 2 megabyte memory board using a total of only 16 chips, the same number of chips currently used in a 512K CoCo 3 memory upgrade board. With this in mind, Tony decoded on the "CPU board" of his 1 meg upgrade BOTH of the two "missing" GIME chip memory manager bits. He used only ONE of those two bits (the low order one) for decoding the 1 megabyte of memory his product was speced to provide, but his CPU board DOES decode the second (highest order) missing MMU bit. Over time, Tony realized that there were NO one megabit DRAM parts made that could be refreshed with the GIME chip's 256 cycle refresh. Whether by mistake or design, Tandy in its layout of the GIME chip had effectively prevented the use of high density DRAM chips with 1 or more megabits of memory per chip.

Faced with this reality, Tony decided that it would be foolish to attempt a 2 meg upgrade. A 2 meg upgrade would require a total of SIXTY FOUR 256K by 1 chips. That means 48 more loads on each address line buffer of the CoCo 3 than was originally planned by the designers! Apart from the physical problem of where to PUT all those chips, one was faced with the near certainty that the address and data buffers of the CoCo 3 would not be able to handle that much fan out. Tony wisely decided that 1 meg was enough.

Enter the Mad Hacker:

There's always at least one crazy person who, upon being told something is impossible, will bend over backwards to show it is not. There may have been one or two BEFORE the story I am about to tell you of, but I am familiar only with this one story. While there may have been previous successful Coco 3 2 meg upgrades, this one is significant because it has been widely and explicitly publicized thru Delphi's OS9 and CoCo SIGs (THE place to be for CoCo 3 hardware news), and provoked within days of its announcement two other successful 2 meg upgrades to be done.

I told my friend Dennis McMillian (COCOKIWI on Delphi) about the fact that the second bit was decoded. I also told him exactly how the decoding of the CAS line (that is used as a 512K bank select) is done on the Disto CoCo 3 1 meg memory upgrade board. I told him, however, that he would be foolish to try to do a 2 meg upgrade, because of the "fan out" problem I spoke of above. Dennis set out to prove me wrong.

Dennis noticed that adding sixteen more chips (as in the official 1 meg upgrade) did NOT result in a fan out problem. Therefore, empirically, he could rely on a fan out for a total of 32 chips. He then checked out specifications for TTL logic chips, and noticed that the rated output fanout for 74F series chips was 2.5 times that rated for 74LS series chips.

Dennis proceeded to destructively remove FOUR twenty pin IC's from the CoCo 3's main board: IC10, 11, 12, and 13. These are one LS374 and three LS244 chips. He then soldered on sockets where those chips used to be, and inserted F244 and F374 chips in place of the LS series chips. Now his CoCo's data and address buffers were not only faster (which is of help in getting around the delays introduced by the 1 meg upgrade board) but also had 2.5 times the output drive, and so could reliably drive over twice the number of memory chips.

We found that the 74F139 chip on the Disto 1 meg board was used to do the bank selection. It had the input pins 2 and 3 shorted together. You need to BREAK the short between those two pins, then send the low order bank select line (the line from the CPU board that formerly went to J5 on the memory board) to pin 2, and the high order bank select line from the Disto "CPU board" to pin 3 of the F139 on the Disto 1 meg memory board.

The high order bank select line is located on the Disto "CPU board" on the four pin connector J2. On that board, as you go from the end of that connector labelled "J2" to the end near the word "OUT", the pins are as follows: +5 volts, lower order bank select bit (used with normal 1 meg upgrades), ground, higher order bank select bit (used only with 2 meg upgrades).

Now that you have re-wired the INPUTS to the bank select chip for 2 meg operation you must wire up the extra 32 memory chips. THIS is a delicate operation. You must PIGGY BACK the additional 32 chips ON TOP OF the existing 32 chips that currently populate the two boards in your one meg upgrade. Alternatively, you can do "4 stack" and make quadruple piggy backs of memory chips, and then use ONLY the Disto one meg board. In any case, each extra layer of chips must have pin 15 (CAS, used as a bank select) bent out. And have those pin 15's all wired to each other for each 16 chip bank. Thus, you have added two new CAS bank select lines. One goes to pin 6 of the F139 chip on the disto memory board, and the other goes to pin 5.

Note that the first 512K will be selected by pin 4 of the F139, and will be found in the PHYSICALLY LOWER bank of chips on the Disto memory board. Pin 5 selects the next 512K bank, and pin 6 selects the bank after that. Pin 7 is currently wired so that it goes to the physically lower bank of chips (original bank of chips) on the memory board that is plugged IN to the Disto memory board.

Users on Internet and Delphi OS9 SIG can tell you how to patch the OS9 kernel so that it will recognise all 2 megs of memory.

This procedure is VERY tedious, highly experimental, and subject to numerous ways it can fail. It requires VERY good soldering and desoldering technique, and should NEVER be attempted by ANY except the MOST experienced hardware hackers! However, I now have THREE reports of successful 2 meg upgrades.

Other Hints: DO NOT USE other than 120 or 150 ns rated DRAMs. Faster chips (80 or 100 ns) have been reported to NOT WORK in either one or two meg upgrades. Bruce Isted and Dave Myers (CoCoPRO!) have both told me about this. You may wish to use NEC brand chips. You MAY wish to try using CMOS instead of NMOS DRAMs to hold down power consumption. Bruce Isted suggests, if you have problems, you may wish to use different memory chips, and you may wish to try shorting out the 120 ohm CAS resistors on the CoCo 3 board, and instead inserting 120 ohm CAS resistors for each bank of memory between the relevant pin of the F139 chip and the CAS lines on each bank of memory chips.

The best source I know of for CHEEEP memory chips is an electronic chip SALVAGE company called FOX Electronics. 120 and 150 ns SOLDER PULL (be sure to specify long pins if possible!) 41256 DRAMs sell for about 50 cents each there. They can be reached at 408-943-1577, or at 2558 Seaboard Ave, San Jose, CA 95131. Ask for "Woody" and tell him Marty Goodman sent ya. They have good prices on other chips, too, when available: 1 meg by 1 DRAMs 100 ns for $4.00 each, 386-20's for $70 each, 4 by 256K 80 ns DRAMs for $4.50 each, 27C1024 32 pin "1megabit" (8 by 128K) CMOS EPROMs for $4.50 each. Prices and availability and chip brands will vary from day to day, so CALL before you order!

I personally regard this upgrade as a HIGHLY impractical, tedious, and potentially dangerous and flakey thing to do. However, as I noted, at least three folks have successfully done it, and CLAIM their computers are running completely reliably.

Given how experimental this procedure is, PLEASE report your successes or failures to CoCo users at large via the Internet Mailing List or Delphi's OS9 or CoCo SIG. I can be reached on Delphi as username MARTYGOODMAN and via Internet at address MARTYGOODMAN@BIOTECHNET.COM.

---marty (Goodman)

**FROM THE LIBRARIAN**

If you have no idea were the Clock module in your CoCo Level 2 OS9Boot file comes from, maybe this will help. By the way, has anyone got the time? Happy computing! Jean-Pierre.

| Module | CRC | Origin | Comments |
|---|---|---|---|
| clock | D28AFD | OS9 Level II System Disk | 60 Hz |
| clock | DDFD68 | OS9 Level II System Disk | 50 Hz |
| clock | 3870A9 | Public Domain Library #11 | serialmouse 60 Hz |
| clock | 3720AE | Public Domain Library #11 | serialmouse 50 Hz |
| clock | 0AAA65 | Disto_4inl | 6242.slot1 |
| clock | D7C0C4 | Disto_4inl | 6242.slot2 |
| clock | 307F44 | Disto_4inl | 6242.slot3 |
| clock | ED15E5 | Disto_4inl | 6242.slot4 |
| clock | | Clock9 | soft.60hz |
| clock | | Clock9 | dsto2.60hz |
| clock | | Clock9 | dsto2.s4.60hz |
| clock | | Clock9 | dsto4.60hz |
| clock | | Clock9 | bb.s3.60hz |
| clock | | Clock9 | bb.s2.60hz |
| clock | ACF042 | Clock9 | soft.50hz |
| clock | | Clock9 | dsto2.50hz |
| clock | | Clock9 | dsto2.s4.50hz |
| clock | 72A150 | Clock9 | dsto4.50hz |
| clock | | Clock9 | dsto4.s4.50hz |
| clock | | Clock9 | bb.s3.50hz |
| clock | | Clock9 | bb.s2.50hz |
| clock | | OCN_OS9SYS | Clockbb: bbl |
| clock | DF89F9 | Public Domain ? | software clock ? |

CHRISTMAS
GREETINGS

and best
wishes

for
Happiness

in the

NEW YEAR